Recognition for acyclic context-sensitive grammars is probably polynomial for fixed grammars

Aarts, E.

Publication date:
1991

Document Version
Publisher's PDF, also known as Version of record

[redacted]

ITK

MEMO

[redacted]

INSTITUTE FOR LANGUAGE TECHNOLOGY AND ARTIFICIAL INTELLIGENCE INSTITUUT VOOR TAAL- EN KENNISTECHNOLOGIE

I.T.K. Research Memo
June 1991

Recognition for Acyclic Context-Sensitive Grammars is probably Polynomial for Fixed Grammars

Erik Aarts

no. 8

Recognition for Acyclic Context-Sensitive Grammars is Probably Polynomial for Fixed Grammars

Erik Aarts
Institute for Language Technology and Artificial Intelligence
PoBox 90153
5000 LE Tilburg
The Netherlands

Abstract

Context-sensitive grammars in which each rule is of the form $\alpha Z \beta \rightarrow \alpha\gamma\beta$ are acyclic if the associated context-free grammar with the rules $Z \rightarrow \gamma$ is acyclic. The problem whether an inputstring is in the language generated by an acyclic context-sensitive grammar is probably polynomial for fixed grammars.

Introduction

One of the most well-known classifications of rewrite grammars is the Chomsky hierarchy. Grammars and languages are of type 0 (unrestricted), type 1 (context-sensitive), type 2 (context-free) or of type 3 (regular). Much research has been done involving regular and context-free grammars. Context-free languages can be recognized in a time that is polynomial in the length of the input and the length of the grammar [Earley, 1970]. Recognition of type 0 languages is undecidable. We see two majors tracks for the research on grammars which lie between these two very distant complexity classes.

First, people have tried to put restrictions on context-sensitive grammars in order to generate context-free languages. Among them are Book [1972], Hibbard [1974] and Ginsburg and Greibach [1966]. Baker [1974] has shown that these attacks come down to the same more or less. They all block the use of context to pass information through the string. Book [1973] gives an overview of attempts to generate context-free languages with non-context-free grammars. How to restrict permutative grammars in order to generate context-free languages is described in Mäkkinen [1985].

The other track is the track of complexity of recognition. One of the best introductions to complexity theory is Garey and Johnson [1979]. They state that recognition for context-sensitive grammars is PSPACE-complete (referring to [Kuroda, 1964] and [Karp, 1972]). Some people have tried to put restrictions on CSG's so that recognition lies somewhere between PSPACE and $\mathcal{P}$. Book [1978] has shown that for *linear time* CSG's recognition is NP-complete even for (some) fixed grammars. Furthermore there is a result that recognition for *growing* CSG's is polynomial for fixed grammars [Dahlhaus and Warmuth, 1986]. This is the tradition I am following.

In this article I will consider one type of restricted context-sensitive grammars, the *acyclic* context sensitive grammars. The complexity of recognition is lower than in the unrestricted case because we restrict the *amount* of information that can be sent (and we do not block it by barriers!). In the unrestricted case we can send messages that *leave no trace*. After a message that changes 0's into 1's e.g. we can send a message that does the reverse. In sending a message from one position in the sentence to another, the intermediate symbols are not changed. In fact they are changed twice: back and forth. With acyclic csg's, this is not possible and the amount of information that can be sent is restricted by the grammar.

Definitions

A *grammar* is a 4-tuple, $G = (V, \Sigma, R, S)$, where
V is a set of symbols, $\Sigma \subset V$ is the set of terminal symbols.
$R \subset V^* \times V^*$ is a relation defined on strings. Elements of R are called rules. $S \in V$ is the startsymbol.

A grammar is *context-sensitive* if each rule is of the form
$\alpha Z \beta \rightarrow \alpha \gamma \beta$ where $Z \in V \setminus \Sigma$; $\alpha, \beta, \gamma \in V^*$; $\gamma \neq e$.
A grammar is *context-free* if each rule is of the form
$Z \rightarrow \gamma$ where $Z \in V \setminus \Sigma$; $\gamma \in V^*$; $\gamma \neq e$.

Derivability ($\Rightarrow$) between strings is defined as follows:
$u\alpha v \Rightarrow u\beta v$ $(u, v, \alpha, \beta \in V^*)$ iff $(\alpha, \beta) \in R$.
The transitive closure of $\Rightarrow$ is denoted by $\stackrel{+}{\Rightarrow}$. The transitive reflexive closure of $\Rightarrow$ is denoted by $\stackrel{*}{\Rightarrow}$. The *language* generated by G is defined as
$L(G) = \{w \in \Sigma^* \mid S \stackrel{*}{\Rightarrow} w\}$.

A *derivation* of a string δ is a sequence of strings $x_1, x_2, \ldots, x_n$ with
$S = x_1$, for all i $(1 \leq i < n)$ $x_i \Rightarrow x_{i+1}$ and $x_n = \delta$.

A context-free grammar is *acyclic* if there is no $Z \in V \setminus \Sigma$ such that
$Z \stackrel{+}{\Rightarrow} Z$. This implies that there is no string $\alpha \in V^*$ such that $\alpha \stackrel{+}{\Rightarrow} \alpha$.

We can map a context-sensitive grammar G onto its *associated* context-free grammar G' as follows: If G is (V, Σ, R, S) then G' is (V, Σ, R', S) where for every rule $\alpha Z \beta \rightarrow \alpha \gamma \beta \in R$ there is a rule $Z \rightarrow \gamma \in R'$. There are no other rules in R'.

We call G *acyclic* iff the associated context-free grammar G' is acyclic.

The notation we use for context-sensitive rules is as follows: the rule
$\alpha Z \beta \rightarrow \alpha \gamma \beta$ is written as $Z \rightarrow [\alpha_1][\alpha_2]\ldots[\alpha_i]\ \gamma\ [\beta_1][\beta_2]\ldots[\beta_j]$ with
$\alpha = [\alpha_1][\alpha_2]\ldots[\alpha_i]$ and $\beta = [\beta_1][\beta_2]\ldots[\beta_j]$.

Recognition is polynomial

In this section we try to prove that the recognition problem for acyclic context-sensitive grammars is polynomial. Acyclic CSG will be abbreviated as ACSG. Suppose we have an acyclic context-sensitive grammar $G = (V, \Sigma, R, S)$.

RECOGNITION FOR A FIXED ACYCLIC CSG
INSTANCE: a string $w \in \Sigma^*$.
QUESTION: Is w in the language generated by G ?

We try to prove that RECOGNITION FOR A FIXED ACYCLIC CSG is polynomial for every ACSG. We shall give an algorithm that recognizes sentences of some ACSG. We shall show that the algorithm is polynomial for some "hard" combinations of grammars and inputs. We are not yet able to prove that the algorithm is polynomial for all grammars and inputs.

A (standard) algorithm for recognition with context-free grammars

In this section I will give a simple algorithm for recognition of sentences generated by a context-free grammar. The algorithm is based on the algorithms for chart parsing that are described in Gazdar and Mellish [1989, chap. 6]. The origin of all these algorithms is Earley's algorithm [Earley, 1970]. Chart parsing is also described in Winograd [1983, pp. 116-127].

The basic datastructure that is used in the recognizer is the *edge*.
An edge E is a 4-tuple, $E = (V_1, V_2, M, T)$, where
V_1 and V_2 are integers, $M \in V \setminus \Sigma$ is a symbol (the mother).
$T \in V^*$ is a list of symbols (the remainder). T is the list of daughters that E expects.

Maybe it is good to say that this is a little different from the edges that Gazdar and Mellish use. Because we are not parsing but recognizing sentences, we are not interested in what has been found yet but only in what has to be found.

Edges are in fact partial results. In order to apply a rule bottom-up we must have found all its daughters. An edge is an element that says that we have found some daughters of a rule but not all of them. Edges are called inactive when the remainder is empty, otherwise they are active. The algorithm uses a chart and an agenda. Both consist of active and inactive edges.

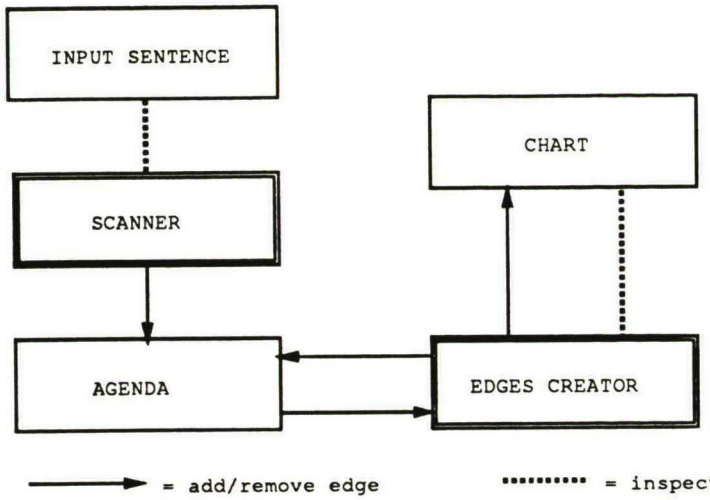
The algorithm we give is a bottom-up algorithm. It consists of two parts: the scanner and the creator of new edges. The scanner reads one input word and does a lexicon lookup. The result is a set of edges. The edges are put in the agenda. Then the creator of new edges is started. The creator of new edges moves an edge from the agenda to the chart and if the remainder is empty it applies two (meta-) rules on it: the Bottom-up Rule and the Fundamental Rule.

> **The Bottom-up rule**
> If you are adding edge $< i, j, A, [\,] >$ to the chart, then for every rule in the grammar of the form $B \rightarrow A\ W$, add an edge $< i, j, B, W >$ to the agenda[1].

> **The Fundamental rule**
> If you are adding edge $< j, k, B, [\,] >$ to the chart, then for every edge in the chart of the form $< i, j, A, [B|W] >$, add a new edge in the agenda of the form $< i, k, A, W >$.

Both the agenda and the chart can be seen as sets. They need not be ordered and the creator of new edges can take an arbitrary edge from the agenda in order to move it to the chart. In the algorithm, the chart and the agenda are represented as lists. When we append the newly created edges at the back of the agenda, we get some kind of breadth-first behaviour. If we append it on the front, the result is a depth-first behaviour.

[1] W is a (possibly empty) list of symbols. A and B are symbols.



When the agenda does not contain edges any more, the creator of new edges gives back control to the scanner. The scanner reads the next input word, fills the agenda and new edges can be created again. When there are no input words left anymore, the recognition is almost finished. The algorithm only has to look whether the chart contains an inactive edge with the startsymbol as its mother and which spans the complete input. Appendix A contains the Prolog-code of the algorithm.

The basic algorithm for ACSG

This algorithm is very similar to the one in the previous section. The definition of an edge is slightly different:

An edge E is a 4-tuple, $E = (V_1, V_2, M, T)$, where
V_1 and V_2 are integers, $M \in V^*$ is a list of symbols (the to-add-list).
$T \in V^*$ is a list of symbols (the remainder). T is the list of symbols that E expects.

Only M has been changed. In a context-free grammar, the lefthand-side of a rule is always a single constituent. But now we have rules of the form $\alpha Z \beta \rightarrow \alpha\gamma\beta$. The lefthand-side is a list of symbols ($\alpha Z \beta$) that has to be added in the chart when the daughters ($\alpha\gamma\beta$) have been found. When M is a list containing one symbol, we call the edge context-free (and otherwise context-sensitive).

The architecture of the system as depicted in Figure 1 remains the same. The creator of new edges takes an edge from the agenda. If it is active, the edge is moved to the chart and the creator of new edges can take the next edge from the agenda. If the edge is context-free and inactive, the edge is moved to the chart and the creator of new edges starts applying the meta-rules:

The Bottom-up rule
If you are adding edge $< i, j, [A], [\,] >$ to the chart, then for every rule in the grammar of the form $W_1 \rightarrow A\ W_2$, add an edge $< i, j, W_1, W_2 >$ to the agenda[2].

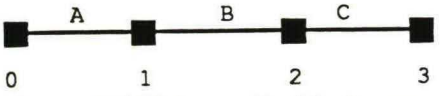
The Fundamental rule
If you are adding edge $< j, k, [B], [\,] >$ to the chart, then for every edge in the chart of the form $< i, j, W_1, [B|W_2] >$, add a new edge in the agenda of the form $< i, k, W_1, W_2 >$.

But what if the edge is context-sensitive and inactive? The creator of new edges has to apply the Split rule now.

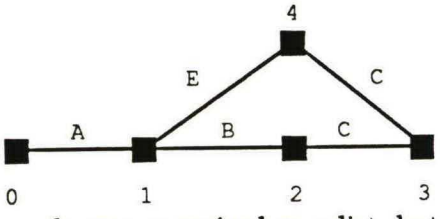
[2] W_i are lists of symbols. A and B are symbols.

The Split rule
Take the edge $< i, j, W_1, [\,] >$ from the agenda and put a number of edges back in the agenda. These edges connect *new vertices* and the symbols one sees walking along the *new path* are exactly the symbols in the lefthandside of the applied grammar rule.

This can be made clear with an example. Suppose we have read some input words and we have this chart:



Now we found that the rule $E \rightarrow B\ [C]$ is applicable because we have the inactive context-sensitive edge $< 1, 3, [E, C], [\,] >$. The Split rule says we have to put the edges $< 1, 4, [E], [\,] >$ and $< 4, 3, [C], [\,] >$ in the agenda. Vertex 4 is a new vertex.



In the previous section the agenda was organized as a list, but the creator of new edges could take an arbitrary edge from it (thanks to the fact that all edges in the agenda had the same end vertex). In this algorithm, we have to be more careful. It is obvious that the edge $< 1, 4, [E], [\,] >$ should be used earlier to generate new edges than $< 4, 3, [C], [\,] >$. The edges in the agenda must be *ordered* ! It is not very complex to keep the right order in the agenda however: if we apply the Split rule, we have to append the list of new edges *in front of* the agenda. The agenda is a FIFO-stack.

When the scanner has read all input words, we have to look again in the chart whether there is an inactive edge with the startsymbol as its mother spanning the entire input. The Prolog-code of this algorithm is also in the Appendix, namely in Appendix B.

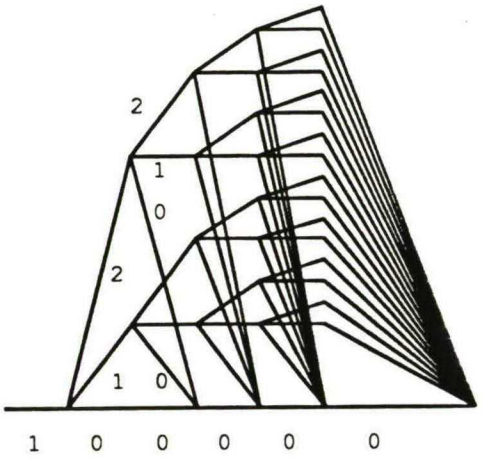
Problematic grammars

In the previous section a straightforward and simple algorithm has been described. The question is whether this algorithm is polynomial and the answer is no.

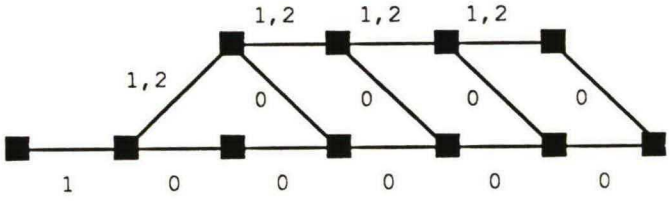
Consider the grammar:

1 → [1] 0
1 → [2] 0
2 → [1] 0
2 → [2] 0
and the input: 100000

After every scan the number of vertices that has to be added is doubled.



Obviously the algorithm is not polynomial. This chart, however, is equivalent with the chart:



Both charts contain the same paths. The algorithm we used is too simple and we have to design an algorithm that "collapses edges" when possible.

Refinements of the basic algorithm

The source code of the refined algorithm is in Appendix C. It is an extension of the previous algorithm. I shall describe the changes I made one by one:

- Left-hand-sides in edges are packed. As a result, there is always at most one complete edge between two points in the agenda and the chart.
- The agenda is treated as a LIFO-stack. New edges are put on the back. They are taken from the front. We have to be careful with this: Edges that start in, say, V1 have to wait on processing of edges that end in V1. We have to search the agenda till we find the first edge without "predecessor" (fwop). We do not do this when there is already a vertex with the same outgoing "set of paths".
- The lefthandsides of edges are packed. Therefore, we have to process a list of list of symbols that have to be added. This is the point where we have to do some collapsing. We search for the longest list in the list of lists. The first symbol of this longest list is the symbol we are going to add. If necessary, we create a new vertex and we shift the symbol from all the list starting with that symbol.
- I also implemented a graphic output of the parser which is easier to read than the output of `redraw`. People who are interested in it can receive it.

Conclusions

The question is whether the refined algorithm is polynomial. Unfortunately, I have not been able to prove this. I can not think of an invariant restricting the size of the agenda or the chart. In one or another way, we should use the acyclicity of the grammar rules.

References

Baker, B. S., Non-context-Free Grammars Generating Context-Free Languages, *Inform. and Control*, *24*, 231–246, 1974.

Book, R. V., Terminal context in context-sensitive grammars, *SIAM J. Comput.*, *1*, 20–30, 1972.

Book, R. V., On the Structure of Context-Sensitive Grammars, *Internat. J. Comput. Inform. Sci.*, *2*, 129–139, 1973.

Book, R. V., On the Complexity of Formal Grammars, *Acta Inform.*, *9*, 171–181, 1978.

Dahlhaus, E. and M. K. Warmuth, Membership for Growing Context-Sensitive Grammars Is Polynomial, *Internat. J. Comput. Inform. Sci.*, *33*, 456–472, 1986.

Earley, J., An Efficient Context-Free Parsing Algorithm, *Comm. ACM*, *13*(2), 94–102, Feb. 1970.

Garey, M. R. and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*, W. H. Freeman and Company, San Francisco, CA, 1979.

Gazdar, G. and C. Mellish, *Natural Language Processing in Prolog*, Addison Wesley, Reading, MA, 1989.

Ginsburg, S. and S. A. Greibach, Mappings which Preserve Context Sensitive Languages, *Inform. and Control*, *9*, 563–582, 1966.

Hibbard, T. N., Context-Limited Grammars, *J. Assoc. Comput. Mach.*, *21*(3), 446–453, July 1974.

Karp, R. M., Reducibility among combinatorial problems, in *Complexity of Computer Computations*, edited by R. E. Miller and J. W. Thatcher, pp. 85–103, Plenum Press, New York, 1972.

Kuroda, S. -Y., Classes of Languages and Linear-Bounded Automata, *Inform. and Control*, *7*, 207–223, 1964.

Mäkkinen, E., On Permutative Grammars Generating Context-Free Languages, *BIT*, *25*, 604–610, 1985.

Winograd, T., *Language as a cognitive process: syntax*, Addison Wesley, Reading, MA, 1983.

Appendix A: Context free chart recognition

```
recognize(String) :-
        scan(String,0,[],Chart,Vend),
        initial(Startsymbol),
        member(edge(0,Vend,Startsymbol,[]),Chart).

scan([],Vend,Finalchart,Finalchart,Vend).
scan([Word|Words],V0,Chartin,Finalchart,Vend) :-
        V1 is V0 + 1,
        findall(edge(V0,V1,Category,[]),      % lexicon lookup
                word(Category,Word),
                Agenda),
        extend_edges(Agenda,Chartin,Chartout),
        scan(Words,V1,Chartout,Finalchart,Vend).

extend_edges([],Finalchart,Finalchart).
extend_edges([Edge|Agenda],Chart,Finalchart) :-
        member(Edge,Chart),!,
        extend_edges(Agenda,Chart,Finalchart).
extend_edges([Edge|Agenda],Chart,Finalchart) :-
        create_new_edges(Edge,Chart,Edges),
%       add_edges(Agenda,Edges,New_agenda),   % breadth-first processing
        add_edges(Edges,Agenda,New_agenda),   % depth-first processing
        extend_edges(New_agenda,[Edge|Chart],Finalchart).

create_new_edges(edge(V1,V2,Category1,[]),Chart,Edges) :-
        findall(edge(V1,V2,Mother1,Tofind1),  % Bottom-up rule
                rule(Mother1,[Category1|Tofind1]),
                Edges1),
        findall(edge(V0,V2,Mother2,Tofind2),  % Fundamental rule
                member(edge(V0,V1,Mother2,[Category1|Tofind2]),Chart),
                Edges2),
        append(Edges1,Edges2,Edges).
create_new_edges(edge(V1,V2,Category1,[Category2|Tofind]),Chart,[]).

add_edges([],Edges,Edges).        % add_edges = add unless it is a member
add_edges([Edge|Edges],Edges1,Edges2) :-
        member(Edge,Edges1),!,
        add_edges(Edges,Edges1,Edges2).
add_edges([Edge|Edges],Edges1,[Edge|Edges2]) :-
        add_edges(Edges,Edges1,Edges2).

findall(Item,Goal,Items) :-
        bagof(Item,Goal,Items),!. % built-in predicate, collects Items
findall(I,G,[]).                                  % satisfying Goal

% rule(s,[np,vp]).  word(np,kim).  initial(s).
% example rules, lexicon, startsymbol
```

Appendix B: Acyclic Context Sensitive chart recognition

```
recognize(String) :-
        retractall(hvn(_)),    % hvn = Highest Vertex Number
        assert(hvn(0)),
        scan(String,0,[],Vend,Chart),
        initial(Startsymbol),
        member(edge(0,Vend,[Startsymbol],[]),Chart).

scan([],Vend,Finalchart,Vend,Finalchart).
scan([Word|Words],V0,Chartin,Vend,Finalchart) :-
        freshnumber(V1),
        findall(edge(V0,V1,[Category],[]),
                word(Category,Word),
                Agenda),
        extend_edges(Agenda,Chartin,Chartout),
        scan(Words,V1,Chartout,Vend,Finalchart).

extend_edges([],Finalchart,Finalchart).
extend_edges([Edge|Agenda],Chart,Finalchart) :-
        member(Edge,Chart),!,
        extend_edges(Agenda,Chart,Finalchart).
extend_edges([Edge|Agenda],Chart,Finalchart) :-
        Edge = edge(V0,V1,[Category],[]),!,     % inactive, context-free
        create_new_edges(Edge,Chart,Edges),
        add_edges(Edges,Agenda,New_agenda),
        extend_edges(New_agenda,[Edge|Chart],Finalchart).
extend_edges([Edge|Agenda],Chart,Finalchart) :-
        Edge = edge(V0,V1,Toadd,[]),            % inactive, context-sensitive
        split(V0,V1,Toadd,Edges),
        add_edges(Edges,Agenda,New_agenda),
        extend_edges(New_agenda,Chart,Finalchart).
extend_edges([Edge|Agenda],Chart,Finalchart) :-
        Edge = edge(_,_,_,[Tofind|Rest]),       % active
        extend_edges(Agenda,[Edge|Chart],Finalchart).

split(V0,V1,[Category],[edge(V0,V1,[Category],[])]).
split(V0,V1,[Category1,Category2|Rest],[Edge|Edges]) :-
        freshnumber(V2),
        Edge = edge(V0,V2,[Category1],[]),
        split(V2,V1,[Category2|Rest],Edges).
```

```
freshnumber(J) :-          % generate fresh vertex number
        hvn(I),
        J is I + 1,
        retract(hvn(I)),
        assert(hvn(J)),!.

create_new_edges(edge(V1,V2,[Category],[]),Chart,Edges) :-
        findall(edge(V1,V2,Toadd1,Tofind1),     % Bottom-up rule
                init(Category,Toadd1,Tofind1),
                Edges1),
        findall(edge(V0,V2,Toadd2,Tofind2),     % Fundamental rule
                member(edge(V0,V1,Toadd2,[Category|Tofind2]),Chart),
                Edges2),
        append(Edges1,Edges2,Edges).

init(Category,Toadd,Tofind) :-
        rule(Lhs,Context_left,Rhs,Context_right),
        append3(Context_left,Rhs,Context_right,[Category|Tofind]),
        append3(Context_left,[Lhs],Context_right,Toadd).

% append3 appends 3 lists (trivial)

% rule(s,[],[np,vp],[]).  word(np,kim).  initial(s).
```

Appendix C: Refined Acyclic Context Sensitive chart recognition

```
recognize(String) :-
        retractall(hvn(_)),
        assert(hvn(0)),
        next_word(String,0,[],Chart,_),!,
        write('I start drawing !'),nl,
        redraw(Chart).

next_word([],Vend,Finalchart,Finalchart,Vend).
next_word([Word|Words],V0,Chartin,Finalchart,Vend) :-
        freshnumber(V1),
        findall(edge(V0,V1,[[Category]],[]),
        word(Category,Word),
        Agenda),
        extend_edges(Agenda,Chartin,Chartout),!,
        next_word(Words,V1,Chartout,Finalchart,Vend).

extend_edges([],Finalchart,Finalchart).
extend_edges(Agenda,Chart,Chart3) :-
        fwop(Agenda,Agenda2,F,Agenda),
        extend_edges2(F,Agenda2,Chart,Chart3).

% First element of a list WithOut Predecessor

fwop([edge(V1,V2,Toa,Tof)|Ag1],Ag1,edge(V1,V2,Toa,Tof),Ag4) :-
        \+ member(edge(_,V1,_,_),Ag4).
fwop([edge(V1,V2,Toa,Tof)|Ag1],[edge(V1,V2,Toa,Tof)|Ag2],F,Ag4) :-
        member(edge(_,V1,_,_),Ag4),
        fwop(Ag1,Ag2,F,Ag4).

extend_edges2(Edge,Agenda,Chart,Finalchart):-
        member(Edge,Chart),!,
        extend_edges(Agenda,Chart,Finalchart).
extend_edges2(edge(B,E,Listoflists,[I|I2]),Agenda,Chart,Finalchart) :- !,
        add_edges([edge(B,E,Listoflists,[I|I2])],Chart,Chart2),
        extend_edges(Agenda,Chart2,Finalchart).
extend_edges2(edge(B,E,[[Listoh]],[]),Agenda,Chart,Finalchart) :- !,
        new_edges(edge(B,E,[[Listoh]],[]),Chart,Edges),
        add_edges(Edges,Agenda,Newagenda),
        extend_edges(Newagenda,[edge(B,E,[[Listoh]],[])|Chart],Finalchart).
extend_edges2(edge(B,E,Listoflists,[]),Agenda,Chart,Finalchart) :-
        splitgraph(edge(B,E,Listoflists,[]),Listofedges,Agenda),
        append(Listofedges,Agenda,Newagenda),
        extend_edges(Newagenda,Chart,Finalchart).
```

```
splitgraph(edge(B,E,Lol,[]),Listofedges,Agenda) :- % Loe max 3 BK KE BE
        searchlongest(Lol,[F|Uit],[],0),
        Uit \== [],!,
        extract(F,Lol,Lol2,Lol3),                    % 2 K-E, 3 B-E
        sort(Lol2,Lol2s),sort(Lol3,Lol3s),
        checkpack(B,E,F,Lol2s,Lol3s,Agenda,Listofedges).
splitgraph(edge(B,E,Lol,[]),Listofedges,_) :-
        singlestosingles(B,E,Lol,Listofedges).

extract(_,[],[],[]).
extract(F,[[F|R1]|R2],[R1|Uitlist],K) :-
        R1 \== [],!,
        extract(F,R2,Uitlist,K).
extract(F,[[F2|R1]|R2],K,[[F2|R1]|Uitlist]) :-
        extract(F,R2,K,Uitlist).

checkpack(B,E,F,Lol2,Lol3,Agenda,[edge(B,E,Lol3,[]),edge(B,K2,[[F]],[])]) :-
        Lol3 \== [],
        member(edge(K2,E,Lol2,[]),Agenda).
checkpack(B,E,F,Lol2,[],Agenda,[edge(B,K2,[[F]],[])]) :-
        member(edge(K2,E,Lol2,[]),Agenda).
checkpack(B,E,F,Lol2,Lol3,Agenda,
                [edge(B,E,Lol3,[]),edge(B,K,[[F]],[]),edge(K,E,Lol2,[])]) :-
        Lol3 \== [],
        \+ member(edge(_,E,Lol2,[]),Agenda),
        freshnumber(K).
checkpack(B,E,F,Lol2,[],Agenda,
                [edge(B,K,[[F]],[]),edge(K,E,Lol2,[])]) :-
        \+ member(edge(_,E,Lol2,[]),Agenda),
        freshnumber(K).

singlestosingles(_,_,[],[]).
singlestosingles(B,E,[[F]|R2],[edge(B,E,[[F]],[])|Uitlist]) :-
        singlestosingles(B,E,R2,Uitlist).

new_edges(edge(V1,V2,[[Category]],[]),Chart,Edges):-
        findall(edge(V1,V2,[Toadd1],Tofind1),
                init(Category,[Toadd1],Tofind1),
                Edges1),
        findall(edge(V0,V2,Toadd2,Tofind2),
                member(edge(V0,V1,Toadd2,[Category|Tofind2]),Chart),
                Edges2),
        append(Edges1,Edges2,Edges).

init(Category,[Toadd],Tofind) :-
        rule(Lhs,L1,L2,L3),
        append3(L1,L2,L3,[Category|Tofind]),
        append3(L1,[Lhs],L3,Toadd).
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% effect is:    add_edges(A,B,C) :- append(B,A,C)
% second argument (B) has already been packed
% C will be packed again
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

add_edges([],Edges,Edges).
add_edges([Edge|Edges],Edges1,Edges2) :-
        member(Edge,Edges1),!,
        add_edges(Edges,Edges1,Edges2).
add_edges([Edge|Edges],Edges1,Edges2):-
        packlhs(Edge,Edges1,Edges5),!,
        add_edges(Edges,Edges5,Edges2).

packlhs(E,[],[E]).
packlhs(edge(A,B,C,D),[edge(A,B,E,D)|Rest],[edge(A,B,Fs,D)|Rest]) :-
        append(C,E,F),!,
        sort(F,Fs).
packlhs(Edge1,[Edge2|Rest],[Edge2|Rest2]) :-
        packlhs(Edge1,Rest,Rest2).

freshnumber(J) :-
        hvn(I),
        J is I + 1,
        retract(hvn(I)),
        assert(hvn(J)),!.

searchlongest([],Current,Current,_).
searchlongest([A|Rest2],Uit,Current,Currentlength) :-
        length(A,L),
        L =< Currentlength,
        searchlongest(Rest2,Uit,Current,Currentlength).
searchlongest([A|Rest2],Uit,_,Currentlength) :-
        length(A,L),
        L > Currentlength,
        searchlongest(Rest2,Uit,A,L).

redraw([]).
redraw([edge(A,B,C,[])|R]) :- !,
        write(edge(A,B,C,[])),nl,
        redraw(R).
redraw([edge(_,_,_,_)|R]) :-
        redraw(R).
```

ITK: P.O. BOX 90153 5000 LE TILBURG THE NETHERLANDS